Semântica Denotacional Legível e Escalável de Linguagens Imperativas

Guilherme H. S. Santos, Roberto S. Bigonha, and Fabio Tirelo

Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Brasil
guisousa,bigonha@dcc.ufmg.br
ftirelo@google.com
http://www.dcc.ufmg.br/llp

Resumo Apesar de ser um poderoso formalismo para a definição da semântica de linguagens de programação, a semântica denotacional é pouco usada. Isso geralmente é atribuído à dificuldade de se ler e escrever definições semânticas formais. Este trabalho aborda uma nova solução para o problema de escalabilidade e legibilidade da Semântica Denotacional, utilizando uma biblioteca de componentes de semântica denotacional que encapsulam conceitos fundamentais e recorrentes de linguagens de programação imperativas e removem a dependência aparente de contexto das construções nas equações semânticas. Consequentemente, a semântica é definida pelo mapeamento direto entre as construções de uma linguagem e as combinações de componentes que modelam sua semântica de uma forma fácil de ler e modificar.

Keywords: semântica de linguagens de programação, semântica denotacional, modularidade, definições baseadas em componentes

1 As Dificuldades de Escalabilidade

Em geral, a apresentação de uma linguagem em livros e manuais continua sendo feita por meio de uma definição formal da sintaxe baseada em gramáticas livres do contexto e uma definição informal da semântica. Mesmo nos casos em que existe uma definição formal da semântica, ela raramente é utilizada efetivamente para comunicação.

Segundo Mosses (2001), uma das principais razões para o uso reduzido da semântica formal é a dificuldade de se ler e escrever definições semânticas. Enquanto a forma de Backus-Naur (BNF) se estabeleceu como uma notação amplamente utilizada e aceita para a definição formal da sintaxe de linguagens de programação, nenhuma metodologia de definição semântica se popularizou. Isso se deve ao fato de que nenhuma delas se aproxima da simplicidade de formalismos sintáticos, tais como gramáticas livres de contexto.

Na abordagem denotacional, a semântica de uma construção é dada pela combinação da semântica de seus constituintes. No entanto, Tirelo, Bigonha, and Saraiva (2008) observam que a semântica de cada construção depende não

somente de seus constituintes, mas também do contexto em que a construção está inserida. O contexto pode ser dividido em: (i) *antecedentes* da construção; (ii) *destinação* da construção e (iii) *estrutura envolvente*. Os *antecedentes* de uma construção compreendem os efeitos dos passos anteriores sobre a sua semântica e são geralmente propagados por estruturas como estados (*stores*) e ambientes (*environments*). Por exemplo, valores atribuídos às variáveis utilizadas fazem parte dos *antecedentes* de uma expressão. A *destinação* de uma construção consiste no passo seguinte à sua execução e é geralmente representada por continuações. Por exemplo, em uma sequência de comandos, a *destinação* do primeiro comando é a sequência de comandos que o seguem. Por fim, a *estrutura envolvente* consiste no local na árvore de sintaxe em que a construção ocorre. Por exemplo, para se definir a semântica do comando *break* é necessário saber se ele ocorre em um laço ou em um comando *switch*.

Na semântica denotacional, o contexto aparece nas equações semânticas sob a forma de parâmetros extras, ou seja, parâmetros que não denotam constituintes da construção cuja semântica é definida. A correta definição da semântica exige que tais parâmetros sejam passados às denotações dos constituintes da construção e à sua destinação. Essa necessidade de manipulação do contexto cria uma dependência entre equações e domínios semânticos. O que é agravado pela inexistência de mecanismos para abstração da implementação dos domínios semânticos em λ-cálculo. Observada também por diversos autores (Liang, Hudak, and Jones, 1995; Mosses, 1988, 2001; Zhang and Xu, 2004), a carência de modularidade das equações semânticas é a principal limitação da metodologia clássica de semântica denotacional.

Consequentemente, para a inclusão de cada nova construção, alterações podem ser necessárias nas equações semânticas das construções previamente definidas, já que estão intimamente ligadas ao contexto em que se inserem. Casos extremos podem exigir que grande parte das construções sejam redefinidas ao se inserir um novo construto na linguagem. Assim, a legibilidade de definições formais de linguagens de grande porte como C++ e Java é comprometida pela alta complexidade das equações semânticas. Em conjunto, essas limitações tornam a semântica denotacional pouco escalável.

O objetivo principal deste trabalho é prover um método escalável de semântica denotacional, com a remoção da dependência aparente do contexto das construções na redação de suas denotações. O escopo de aplicação do método é limitado às linguagens de programação imperativas, de forma a permitir a escolha de um conjunto coeso de conceitos fundamentais e recorrentes dessas linguagens e ainda assim expressivo o suficiente para definir linguagens inteiras. Os conceitos são modelados por uma biblioteca de componentes semânticos, os quais podem ser usados separadamente e combinados de diversas formas.

2 Semântica Denotacional Baseada em Componentes

Embora os parâmetros extras que correspondem ao contexto das construções devam estar presentes em todas as equações semânticas, poucos são os casos em

que sua presença contribui para a compreensão da semântica definida. Contrariamente, a presença explícita do contexto nas equações torna sua leitura mais difícil, devido ao aumento da complexidade. Na equação semântica da construções *if-then-else*, por exemplo, o *environment* r é mencionado quatro vezes na equação com o único objetivo de passá-lo sem alteração aos constituintes da construção:

$\mathcal{C}[\![\texttt{`if' E `then' C}_1\ \texttt{`else' C}_2]\!]$r c = $\mathcal{R}[\![\texttt{E}]\!]$ r;Bool?;cond($\mathcal{C}[\![\texttt{C}_1]\!]$ r c, $\mathcal{C}[\![\texttt{C}_2]\!]$ r c)

Logo, é desejável que se abstraia o contexto nas equações semânticas sempre que ele não influir diretamente na semântica da construção modelada. Entretanto, dada a complexa manipulação do contexto nas equações semânticas, nem sempre é possível abstraí-lo diretamente.

Esse tipo de problema pode ser resolvido pelo uso de combinadores (Curry and Feys, 1958 apud Hughes, 1982). Por exemplo, um combinador **B** pode ser utilizado para alterar a ordem dos parâmetros de uma função:

$$\mathbf{B}\, f\, x\, y = f\, y\, x$$

e, em conjunto com a redução η, para simplificar fórmulas:

$$\begin{aligned} f\, x\, y &= g\, y\, x \\ f\, x\, y &= \mathbf{B}\, g\, x\, y \\ f &= \mathbf{B}\, g \end{aligned}$$

Pode-se utilizar uma estratégia semelhante para simplificar equações semânticas. Considere a produção A e sua denotação:

$$\begin{aligned} &A \to r_0 B_1 r_1 B_2 \ldots B_n r_n \\ &\mathcal{S} : A \to \text{Contexto} \to \text{Ans} \\ &\mathcal{S}[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!]\, c = g(\mathcal{S}[\![B_1]\!], \mathcal{S}[\![B_2]\!], \ldots, \mathcal{S}[\![B_n]\!], c) \end{aligned}$$

Por meio de um combinador k

$$k\,(x_1, x_2, \ldots, x_n)\, c = g(x_1, x_2, \ldots, x_n, c)$$

é possível encapsular o fluxo das informações de contexto na denotação de A e, por meio de redução η, simplificar a equação semântica:

$$\begin{aligned} \mathcal{S}[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] c &= k(\mathcal{S}[\![B_1]\!], \mathcal{S}[\![B_2]\!], \ldots, \mathcal{S}[\![B_n]\!]) c \\ \mathcal{S}[\![r_0 B_1 r_1 B_2 \ldots B_n r_n]\!] &= k(\mathcal{S}[\![B_1]\!], \mathcal{S}[\![B_2]\!], \ldots, \mathcal{S}[\![B_n]\!]) \end{aligned} \tag{1}$$

O resultado (1) é um mapeamento direto entre a construção sintática abstrata e sua semântica. O combinador k também encapsula detalhes da semântica da construção e se for genérico o suficiente poderá ser reutilizado em outras definições como um componente de semântica denotacional.

Para ilustrar a aplicação dessa estratégia, considere a criação de um combinador específico que encapsula a passagem dos parâmetros de contexto para

as funções $\mathcal{E}[\![\mathtt{E}]\!]$, $\mathcal{C}[\![\mathtt{C}_1]\!]$ e $\mathcal{C}[\![\mathtt{C}_2]\!]$, permitindo a reescrita da equação que define a semântica de *if-then-else* como:

$$\mathcal{C}[\![\texttt{`if'}\ \mathtt{E}\ \texttt{`then'}\ \mathtt{C}_1\ \texttt{`else'}\ \mathtt{C}_2]\!]\ \mathrm{r}\ \mathrm{c} = \mathrm{choose}(\mathcal{E}[\![\mathtt{E}]\!], \mathcal{C}[\![\mathtt{C}_1]\!], \mathcal{C}[\![\mathtt{C}_2]\!])\ \mathrm{r}\ \mathrm{c}$$

Aplicando-se redução η, essa equação torna-se:

$$\mathcal{C}[\![\texttt{`if'}\ \mathtt{E}\ \texttt{`then'}\ \mathtt{C}_1\ \texttt{`else'}\ \mathtt{C}_2]\!] = \mathrm{choose}(\mathcal{E}[\![\mathtt{E}]\!], \mathcal{C}[\![\mathtt{C}_1]\!], \mathcal{C}[\![\mathtt{C}_2]\!])$$

onde

$$\mathrm{ifThen}\ (\mathsf{E}, \mathsf{C}_1, \mathsf{C}_2)\ \mathrm{r}\ \mathrm{c} = \mathsf{E}\ \mathrm{r};\mathrm{cond}(\mathsf{C}_1\ \mathrm{r}\ \mathrm{c}, \mathsf{C}_2\ \mathrm{r}\ \mathrm{c})$$

O combinador criado encapsula o fluxo das informações de contexto, simplificando a leitura do mapeamento semântico. Para isso, encapsula-se a semântica da construção, transformando as denotações de seus constituintes em parâmetros do combinador.

3 Legibilidade via Abstração

O processo de encapsulação do contexto pode ser aplicado às equações semânticas de todas as construções, resultando em um mapeamento semântico mais legível que a definição inicial, como mostrado pela Figura 2. Observe que as equações semânticas utilizam combinadores que denotam comandos, expressões, declarações e programas. A definição de cada um dos combinadores usados na Figura 2 encontra-se nas Tabelas 1 e 2. O fluxo de informações de contexto é encapsulado pelos combinadores, exceto na denotação de programas. Esse é o ponto em que o contexto inicial é definido. A partir dele o contexto flui de acordo com a semântica denotada pelos combinadores. Para entender a definição denotacional da linguagem de exemplo é suficiente a leitura das Figuras 2(b) e 3 e da Tabela 1. Os detalhes da Tabela 2 podem ser abstraídos.

Os combinadores de denotações criados formam uma biblioteca de componentes de semântica denotacional. Conceitos fundamentais de linguagens de programação podem ser encapsulados em componentes, que são utilizados em diversas definições semânticas. Assim, o exercício de definição passa a ser o de combinar componentes previamente definidos de acordo com a semântica pretendida. Essa abordagem foi inspirada na semântica baseada em componentes de Mosses (2005).

Dada a contínua evolução das linguagens de programação, um aspecto importante da biblioteca é a possibilidade de incorporação de conceitos sob a forma de novos componentes. Os esforços são assim voltados para a criação de uma biblioteca abrangente, permitindo a definição semântica de um número significativos de linguagens. Sob essa perspectiva, quanto mais genéricos e abrangentes os componentes, maior a qualidade da biblioteca.

Ilustramos a legibilidade que se pode alcançar com a definição de Small, uma linguagem simples com expressões, declarações e comandos extraída de Gordon (1979), cuja sintaxe abstrata é apresentada na Figura 1.

P → program C
D → var I = E |D_1;D_2
C → E_1 := E_2 |output E |if E then C_1 else C_2 |while E do C
| begin D;C end |C_1;C_2
E → N |true |false |read |I |E_1 O E_2
O → + |- |× |÷ |= |!=

Figura 1: Sintaxe abstrata da linguagem Small.

(a) Semântica Clássica

Programas:
$\mathcal{P}[\![$`program' C`$]\!]$ i = $\mathcal{C}[\![$`C`$]\!]$ r c s[i/'input'][[]/'output']
where r = λi.unbound
c = λs.(stop, s 'output')
s = λl.unused

Declarações:
$\mathcal{D}[\![$`var' I '=' E`$]\!]$ r u = $\mathcal{R}[\![$`E`$]\!]$ r;ref λl.u [l/I]
$\mathcal{D}[\![D_1$`';'`$D_2]\!]$ r u = $\mathcal{D}[\![D_1]\!]$ r;λr_1.$\mathcal{D}[\![D_2]\!]$ r[r_1];λr_2.u (r_1[r_2])

Expressões:
$\mathcal{E}[\![$`'true'`$]\!]$ r k = k true
$\mathcal{E}[\![$`'false'`$]\!]$ r k = k false
$\mathcal{E}[\![$`N`$]\!]$ r k = k $\mathcal{B}[\![$`N`$]\!]$
$\mathcal{E}[\![$`I`$]\!]$ r k = (r I = unbound) →err, k (r I)
$\mathcal{E}[\![$`'read'`$]\!]$ r k s = null (s 'input') →err s,
k (hd (s 'input')) s[tl (s 'input')/'input']
$\mathcal{E}[\![E_1$ `'O'` $E_2]\!]$ r k = $\mathcal{R}[\![E_1]\!]$ r;λe_1.$\mathcal{E}[\![E_2]\!]$ r;λe_2.k ($\mathcal{O}[\![$`O`$]\!]$ e_1 e_2)

Comandos:
$\mathcal{C}[\![E_1$ `':='` $E_2]\!]$ r c = $\mathcal{E}[\![E_1]\!]$ r;Loc?λl.$\mathcal{R}[\![E_2]\!]$ r;update l;c
$\mathcal{C}[\![$`'output' E`$]\!]$ r c = $\mathcal{R}[\![$`E`$]\!]$ r;λe s.c s[[e • s]'output'/'output']
$\mathcal{C}[\![$`'if' E 'then'` C_1 `'else'` $C_2]\!]$ r c =
$\mathcal{R}[\![$`E`$]\!]$ r;Bool?;cond($\mathcal{C}[\![C_1]\!]$ r c, $\mathcal{C}[\![C_2]\!]$ r c)
$\mathcal{C}[\![$`'while' E 'do' C`$]\!]$ =
fix λf r c.$\mathcal{R}[\![$`E`$]\!]$ r;Bool?;cond($\mathcal{C}[\![$`C`$]\!]$ r;f r c, c)
$\mathcal{C}[\![C_1$`';'`$C_2]\!]$ r c = $\mathcal{C}[\![C_1]\!]$ r;$\mathcal{C}[\![C_2]\!]$ r c
$\mathcal{C}[\![$`'begin' C';' D 'end'`$]\!]$ r c = $\mathcal{D}[\![$`D`$]\!]$ r;λr′.$\mathcal{C}[\![$`C`$]\!]$ r[r′] c

(b) Semântica de Componentes

Programas:
$\mathcal{P}[\![$`'program' C`$]\!]$ i = result(**run**($\mathcal{C}[\![$`C`$]\!]$) i r_0 s_0)
where result (t,e,r,s) = (t,s 'output')

Declarações:
$\mathcal{D}[\![$`'var' I '=' E`$]\!]$ = **bind**(`I`, **ref**(**deref**($\mathcal{E}[\![$`E`$]\!]$)))
$\mathcal{D}[\![D_1$`';'`$D_2]\!]$ = **elabSeq**($\mathcal{D}[\![D_1]\!]$, $\mathcal{D}[\![D_2]\!]$)

Expressões:
$\mathcal{E}[\![$`'true'`$]\!]$ = **bool**('true')
$\mathcal{E}[\![$`'false'`$]\!]$ = **bool**('false')
$\mathcal{E}[\![$`N`$]\!]$ = **int**(`N`)
$\mathcal{E}[\![$`I`$]\!]$ = **lookup**(`I`)
$\mathcal{E}[\![$`'read'`$]\!]$ = **input**
$\mathcal{E}[\![E_1$ `'O'` $E_2]\!]$ = **apply**(`O`,$\mathcal{E}[\![E_1]\!]$,$\mathcal{E}[\![E_2]\!]$)

Comandos:
$\mathcal{C}[\![E_1$ `':='` $E_2]\!]$ = **assign**($\mathcal{E}[\![E_1]\!]$, $\mathcal{E}[\![E_2]\!]$)
$\mathcal{C}[\![$`'output' E`$]\!]$ = **output**($\mathcal{E}[\![$`E`$]\!]$)
$\mathcal{C}[\![$`'if' E 'then'` C_1 `'else'` $C_2]\!]$ =
choose($\mathcal{E}[\![$`E`$]\!]$, $\mathcal{C}[\![C_1]\!]$, $\mathcal{C}[\![C_2]\!]$)
$\mathcal{C}[\![$`'while' E 'do' C`$]\!]$ = **loop**($\mathcal{E}[\![$`E`$]\!]$,$\mathcal{C}[\![$`C`$]\!]$)
$\mathcal{C}[\![C_1$`';'`$C_2]\!]$ = **seq**($\mathcal{C}[\![C_1]\!]$,$\mathcal{C}[\![C_2]\!]$)
$\mathcal{C}[\![$`'begin' C';' D 'end'`$]\!]$ =
makeClosure($\mathcal{D}[\![$`D`$]\!]$, $\mathcal{C}[\![$`C`$]\!]$)

Figura 2: Comparação entre semântica denotacional tradicional e abordagem baseada em componentes para a linguagem Small.

Versões da semântica denotacional tradicional e da semântica denotacional baseada em componentes de Small são comparadas na Figura 2. A definição semântica tradicional utiliza como informações de contexto: *environment*, *store* e continuação. Enquanto o contexto está presente em todas as equações semânticas tradicionais, a única equação da definição baseada em componentes que apresenta o contexto explicitamente é a equação semântica de programas. Isso ocorre porque o contexto inicial deve ser definido juntamente com a semântica de programas, mas, a partir desse ponto, o contexto flui implicitamente pelos componentes.

A abordagem baseada em componentes explora o encapsulamento de conceitos fundamentais em componentes. Dessa forma, uma vez que o leitor tenha conhecimento da semântica dos componentes, a leitura da definição se torna mais direta. A remoção da dependência do contexto na redação das equações semânticas e o uso de componentes conferem maior destaque aos conceitos modelados e à forma como eles se combinam.

4 Biblioteca de Componentes

A biblioteca desenvolvida neste trabalho está organizada em três partes: domínios semânticos, componentes de semântica denotacional e funções auxiliares.

Os domínios semânticos (Figura 3) definem os tipos das denotações que representam o modelo computacional característico do paradigma imperativo. Dentre esses domínios, os mais importantes são aqueles que modelam informações de contexto: continuação, *store* e *environment*. A continuação modela o resto do programa, ou seja, as construções que são afetadas pela computação corrente, para as quais o controle flui. O *store* representa um mecanismo abstrato de armazenamento, mapeando endereços chamados de *locations* a valores. Finalmente, o *environment* é o mapeamento entre nomes e valores denotáveis.

A cada domínio que compõe o contexto está associado um domínio de valores. Os valores que podem ser alvo de ligações no *environment* pertencem ao domínio *Dv*, enquanto os valores armazenáveis no *store* pertencem ao domínio *Sv*. Finalmente, os valores que podem ser passados à continuação como resultado de expressões pertencem ao domínio dos valores expressáveis, *Ev*. Esse último domínio abrange comandos, expressões, declarações, arranjos, registros, objetos, classes, procedimentos e valores básicos.

Os valores básicos, representados pelo domínio *Bv*, tem o objetivo de representar valores primitivos: booleanos, *strings*, caracteres e diversos tipos de números. Esse domínio abrange grande parte dos tipos encontrados em linguagens de programação imperativas, mas outros tipos podem ser adicionados à biblioteca caso necessário.

Os principais componentes são representações de programas, comandos, expressões e declarações. Os primeiros, representados pelo domínio *Pd*, modelam programas, lidando com entrada e saída e retornando o resultado da execução. Enquanto a principal função das denotações de comandos, *Cd*, é alterar o estado da máquina, denotações de expressões, *Ed*, são principalmente responsáveis por produzir valores, e denotações de declarações, *Dd*, por produzir ligações. No entanto, essa separação não é exaustiva, havendo componentes que encorporam característica de múltiplos grupos. Esses componentes pertencem ao domínio *Xd*. Finalmente, o domínio *Ex* contempla denotações de expressões e valores, de modo a flexibilizar a ordem de avaliação de expressões. A Tabela 1 apresenta alguns dos principais componentes da biblioteca e breves descrições de sua semântica. A implementação desses componentes está na Tabela 2. Mais detalhes podem ser encontrados na dissertação de mestrado de Guilherme Santos (Santos, 2013b).

Ans	= [{error,stop} × Ev × Env × Store] + [Ev × Env × Store]	- respostas finais
Bv	= Number + Bool + String + Char + {undefined}	- valores básicos
Rv	= Bv + Array + Record + Object	- *R-values*
Dv	= Loc + Rv + Proc + Q + Xd + Scope + Class	- valores denotáveis
Sv	= Rv* + Rv	- valores armazenáveis
Ev	= Dv	- valores expressáveis
Env	= [Id + {type} × Id] →[Dv + {unbound}]	- *environments*
Loc	= [{const,var} × address] + {input,output}	- *locations*
Store	= [{level,new} + Loc] →[Sv + {unused}]	- *stores*
Q	= Ev →Env →Store →Ans	- continuações

Figura 3: Principais domínios semânticos da biblioteca.

Para simplificar as definições dos componentes, são abstraídas em funções auxiliares algumas operações comuns, como a procura de ligações no *environment* e o armazenamento de valores no *store*. Essas funções não precisam ser conhecidas pelo usuário da biblioteca, mas facilitam a criação de novos componentes e facilitam a leitura das definições dos componentes.

Foi desenvolvida uma implementação da biblioteca de componentes na linguagem Haskell e tornada disponível online (Santos, 2013a). Pode-se construir um interpretador mapeando-se a sintaxe abstrata da linguagem aos componentes implementados em Haskell. Dessa forma, aliada a analisadores léxico e sintático, a implementação pode ser utilizada para prototipagem de linguagens de programação.

5 Demonstração de Escalabilidade

Nesta seção é apresentado um estudo de caso para demonstrar a escalabilidade da semântica denotacional baseada em componentes. São definidas três extensões da linguagem Small, utilizando componentes pertencentes à biblioteca apresentada neste trabalho. O impacto dessas extensões é avaliado, e é apresentada a incorporação de um novo conceito à biblioteca de componentes.

5.1 Semântica Denotacional de Small1

A primeira extensão está relacionada à manipulação do fluxo de controle. São adicionados à linguagem Small os sequenciadores *break* e *continue*. O primeiro termina a execução de um laço, e o segundo termina a execução de uma iteração de um laço.

A abordagem de componentes modela diretamente sequenciadores, visto que eles são conceitos fundamentais das linguagens de programação. Dessa forma, além das equações que definem a semântica dos sequenciadores, é necessário

Tabela 1: Exemplos de componentes de semântica denotacional.

run	Cd→File→Env→Store→Ans	**catch**	(Cd, Proc) →Cd
	Modela a execução de um programa e define o contexto inicial. Ao fim da execução, é retornada uma tupla contendo contendo: um código(`stop` ou `error`), um valor e *environment* e *store* correntes.		Instala o procedimento recebido no *environment* para ser utilizado pelo componente `throw` e executa o comando. Caso o procedimento seja executado, sua continuação é a continuação do componente `catch`.
bind	(Id, Xd+Ev) →Dd	**input**	Ed
	Liga um identificador e um valor denotável no *environment*.		Lê um valor da entrada e o passa para a continuação.
deref	Ex →Ed	**assign**	(Ex, Ex) →Cd
	Avalia a expressão e, caso o valor resultante seja um *location*, passa para a continuação o valor armazenado no *store*. Caso contrário, o próprio valor da expressão é passado.		O valor resultante da avaliação da expressão ou variável do primeiro parâmetro é associado no *store* ao *location* resultante da avaliação do segundo parâmetro.
lookup	[Id] →Ed	**elabSeq**	(Dd, Dd) →Dd
	Obtém o valor ligado ao identificador.		Combina declarações sequencialmente.
ref	Ex →Ed	**apply**	((Ev*→Ev+Error), Ex*)→Ed
	Armazena o valor da expressão no *store* e passa o *location* associado para a continuação.		Aplica uma operação definida pelo usuário a uma lista de valores.
pop	Cd	**output**	Ex →Cd
	Remove do *store* os valores associados ao nível corrente da pilha.		Escreve o valor da expressão ou variável no fim da saída.
push	Cd	**seq**	(Cd,Cd) →Cd
	Cria um novo nível na pilha do *store*.		Modela sequenciamento de comandos.
choose	(Ex, x, x) →x, x ∈ Xd	**throw**	Ex →Cd
	Caso o valor da expressão seja verdadeiro, o segundo parâmetro é executado, caso contrário o terceiro parâmetro é executado.		Realiza um salto, passando o valor da expressão para o procedimento instalado no *environment* pelo `catch` mais interno.
escape	(Cd, Id) →Cd	**jump**	Id →Cd
	Instala no *environment* a continuação corrente associando-a ao identificador recebido e executa o comando.		Interrompe o fluxo de controle e executa a continuação associada ao rótulo recebido como parâmetro.
parametrize	(Ad*, Xd) →Proc	**makeClosure**	(Dd, x) →x, x ∈ Ed + Cd
	Realiza abstração de parâmetros. Recebe uma lista de pares que definem os tipo de passagem e o nome dos parâmetros formais e um componente que corresponde ao corpo do procedimento.		Executa o segundo parâmetro utilizando o *environment* recebido acrescido das ligações produzidas pelo primeiro parâmetro. A seguir, todas as alterações feitas ao *environment* pelos dois parâmetros são descartadas.
call	(Ex,Ex*) →Cd	**loop**	(Ex, Cd) →Cd
	Realiza uma chamada de procedimento. O primeiro parâmetro é avaliado para se obter o procedimento e o segundo é uma lista de expressões que constituem os parâmetros reais.		Caso o valor da expressão seja verdadeiro, o comando é executado, e o laço é executado novamente. Caso a expressãp possua valor falso, o controle é transferido para a continuação.
int	String →Ed	**bool**	String →Ed
	Cria um valor inteiro.		Cria um valor booleano.
proc	(Xd + Proc) →Ed		
	Cria procedimentos. O primeiro parâmetro corresponde ao corpo do procedimento.		

Tabela 2: Exemplos de componentes de semântica denotacional.

run : Cd→File→Env→Store→Ans run C i r s = C r q s′ where q = (λe r s.(stop, e, r, s)) s′ = [i,[]/input,output]	**lookup** : [Id+(Ex,Id)+(Id,Id)] →Ed lookup I r q = lookup' I r;λe r′.isnoteval e → eval e r′q, q e r′
bind : (Id, Xd+Ev) →Dd bind(I, X) r q = eval X r;λx r′.q ϕ [f x/I] where f x = isClass x →x['classname'/I], x	**input** : Ed input r q s =null(s input) →err ϕ r s, q hd(s input)r s[tl(s input)/input]
deref : Ex →Ed deref E r q = eval E r;deref';q	**ref** : Ex →Ed ref E r q = eval E r;ref';q
output : Ex →Cd output E r q = eval E r;deref';Rv?; λe r s.q ϕ r s[[e • s output]/output]	**catch** : (Cd, Proc) →Cd catch (C, P) r q = C r[catch′/'catch'] where catch′ e r′q′ = P e r q
parametrize : (Ad*, Xd) →Proc parametrize(A*, X) = (A*, λr q v*.addPar (revert (zip A* v*)) X r q) where addPar a* X′ = null a* →X′, addPar (tl a*)(body(hd a*)X′) body = λ((m,I),e) X″c′.entry m (λe′r′.X″r′[e′/I] (exit m e c′e′))	**apply** : ((Ev*→Ev+Error), Ex*)→Ed apply (O, e*) r q = eval′e* [] op r where eval′e_1^* e_2^* q = null e_1^* →q e_2^*, eval (hd e_1^*);deref'; λv.eval′(tl e_1^*)[e_2^* • v]q op = λe*.Ev?;O e*=error→ err ϕ, q (O e*)
loop : (Ex, Cd) →Cd loop (E_1, C, E_2) = fix λw.choose(eval E_1, seq(C, w), skip)	**choose** : (Ex, x, x) →x, x ∈ Xd choose (E,X_1,X_2) r q = eval E r;deref';Bool?; λb r.cond(eval X_1 r q, eval X_2 r q)b
seq : (Cd,Cd) →Cd seq (C_1,C_2) r q = C_1 r;λe r.C_2 r q	**makeClosure** : (Dd, x) →x, x ∈ Ed + Cd makeClosure(D, X) r q = D r (λe r′.X r′q′) where q′= (λe′r″.q e′r)
elabSeq : (Dd, Dd) →Dd elabSeq(D_1,D_2) r q = D_1 r; λe_1 r_1.D_2 r[r_1]; λe_2 r_2.q ϕ r_1[r_2]	**pop** : Cd pop r q s = (s level) = 1 →err ϕ r s, q ϕ r (λl.l = level →(s level) - 1, l = (_,((s level), _)) →unused,s l)
push : Cd push r q s = q ϕ r (λl.l=level→(s level)+1, s l)	**jump** : Id →Cd jump I r q = lookup' I r;Q?;λq′.q′ϕ
proc : (Xd + Proc) →Ed proc (X) r q = q (λr′.x r) r where x = isXd X →(args, body), x args = [] body = X	**call** : (Ex,Ex*) →Cd call (E_1,E_2) r q = eval E_1;Proc?;λe.e=(a,p)→ evalPar a′E_2(p r q), isXd e →e r q, err ϕ r where a′= hd(unzip a)
throw : (Ex) →Cd throw (I, E) r q =eval E r;λe r′.lookup' 'catch' r′; Q?;λq′.q′e	**assign** : (Ex, Ex) →Cd assign (E_1,E_2) r q = eval E_1 r;Loc?;λl.eval E_2; deref';Rv?update I q

apenas alterar a equação semântica de laços. A equação é organizada em função dos conceitos de repetição e saída, formada pela combinação dos combinadores correspondentes e das denotações dos constituintes da construção. O leitor não precisa conhecer a implementação dos sequenciadores, e o modelo semântico que suporta as equações não é alterado.

Note que a abordagem baseada em componentes não requer alterações dos domínios semânticos, e as equações permanecem legíveis. Isso ocorre porque os sequenciadores são conceitos recorrentes das linguagens de programação impe-

$\mathcal{C}[\![\texttt{while E do C}]\!]$ = escape(loop($\mathcal{E}[\![\texttt{E}]\!]$, escape($\mathcal{C}[\![\texttt{C}]\!]$,'continue')),'break')
$\mathcal{C}[\![\texttt{break}]\!]$ = jump('break')
$\mathcal{C}[\![\texttt{continue}]\!]$ = jump('continue')

Figura 4: Semântica baseada em componentes de sequenciadores de Small1.

rativas e, por isso, são previstos pela biblioteca. Além disso, os componentes são genéricos o suficiente para expressar funcionalidades menos comuns. Por exemplo, o *break* com múltiplos níveis de Java também poderia ser definido pelos mesmos componentes `jump` e `escape`. Bastaria, por meio do componente `escape`, definir saídas identificadas pelos rótulos dos níveis e passar o rótulo do nível desejado como parâmetro do componente `jump`.

5.2 Semântica Denotacional de Small2

Esta seção adiciona procedimentos à linguagem Small1. Para isso, devem ser adicionadas equações que descrevem a semântica da declaração e da chamada dos procedimentos. Além disso, a chamada de procedimentos é responsável por criar uma nova área de memória para as variáveis locais aos procedimentos. Ao fim da chamada, essa área de memória deve ser liberada.

$\mathcal{D}[\![\texttt{I(A);C}]\!]$ = bind(I, proc(parametrize($\mathcal{A}[\![\texttt{A}]\!]$, escape($\mathcal{C}[\![\texttt{C}]\!]$, 'return'))))

$\mathcal{A}[\![\texttt{I}]\!]$ = (value, I)
$\mathcal{A}[\![\texttt{I, A}]\!]$ = [$\mathcal{A}[\![\texttt{I}]\!]$ • $\mathcal{A}[\![\texttt{A}]\!]$]

$\mathcal{C}[\![\texttt{return}]\!]$ = jump('return')
$\mathcal{C}[\![\texttt{E}_1\texttt{(L)}]\!]$ = createMemoryBlock(call($\mathcal{E}[\![\texttt{E}_1]\!]$,$\mathcal{E}[\![\texttt{L}]\!]$))
    where createMemoryBlock p = seq(push, seq(p, pop))

$\mathcal{E}[\![\texttt{E, L}]\!]$ = [$\mathcal{E}[\![\texttt{E}]\!]$ • $\mathcal{E}[\![\texttt{L}]\!]$]

Figura 5: Semântica denotacional baseada em componentes de procedimentos de Small2.

Uma vez que procedimentos e gerência de memória são conceitos fundamentais das linguagens de programação, pode-se utilizar componentes de semântica denotacional na extensão, como mostra a Figura 5. Procedimentos são declarados por meio do componente `bind`, parametrizados por meio de `parametrize` e chamados por meio de `call`. A gerência de memória tem sua semântica modelada pelos componentes `push` – que cria uma nova área de memória – e `pop` – que libera a nova área de memória.

Gerência de memória e procedimentos são funcionalidades complexas, embora fundamentais. Por isso, a definição baseada em componentes permanece legível

apesar da complexidade adicionada à semântica. Note que o impacto da gerência de memória sobre a definição baseada em componentes é localizada, afetando apenas a equação semântica da chamada de procedimentos.

5.3 Semântica Denotacional de Small3

A última extensão adiciona exceções à linguagem Small2. A construção *try-catch-finally* permite especificar que ao executar um comando, as exceções levantadas sejam tratadas por *catches* específicos, e, a seguir, o comando do bloco *finally* seja executado. No entanto, é importante observar que esse comando deve sempre ser executado. Consequentemente, caso encontre-se um retorno do procedimento durante a execução da construção, é necessário executar o bloco *finally* antes de deixar o procedimento.

Exceções:
$\mathcal{C}[\![\texttt{try C}_1\ \texttt{catch I C}_2\ \texttt{finally C}_3]\!]$ = trap(catch(try, exception),
('return', return))
where exception = parametrize((value,I), C_2)
return = seq(C_3, jump('return'))
try = seq(C_1,C_3)
$\mathcal{C}[\![\texttt{throw E}]\!]$ = throw($\mathcal{E}[\![\texttt{E}]\!]$)

Novo Componente:
trap : (Cd, (Id, Cd)*) →Cd
trap (C, L) r q = C r[traps L] q
where traps L = (null L) →[], (hd L) = (i,c) →[c′/i][traps (tl L)]
c′ = λe r′.c r′q

Figura 6: Semântica de exceções baseada em componentes.

Como exceções são conceitos fundamentais de linguagens de programação, sua semântica pode ser definida por meio dos componentes `catch` e `throw`. O código para tratamento da exceção é parametrizado por meio do componente `parametrize`. Note que, como mostra a Figura 6, não é necessário alterar equações previamente definidas. A interceptação do retorno de procedimento é feita utilizando-se o componente `trap` criado pelo usuário da biblioteca. Esta definição permite executar um comando com várias saídas possíveis.

O componente `trap` já existe na biblioteca, entretanto, para exemplificar a criação de um novo componente por parte do usuário, vamos supor que ainda não tivesse sido definido.

A interceptação de desvios do fluxo de controle é uma operação complexa. No entanto, a Figura 6 mostra que esse conceito pode ser facilmente encapsulado em um componente. O Exemplo mostra que conceitos relevantes das linguagens de programação não previstos podem ser encapsulados e incorporados à biblioteca apresentada neste trabalho.

Como pode ser observado nas extensões apresentadas, o encapsulamento de conceitos fundamentais das linguagens de programação em componentes favorece a escalabilidade das definições semânticas. As extensões têm pouco impacto sobre a definição baseada em componentes. Os componentes apresentados neste trabalho são genéricos o suficiente para expressar a semântica de uma gama significativa de construções.

5.4 Trabalhos Relacionados

A semântica incremental (Tirelo et al., 2008) baseia-se no conceito de vagueza nas definições semânticas. Detalhes são acrescentados de forma incremental a uma definição incompleta da semântica de uma linguagem. Os incrementos são compostos por transformações das denotações. Com inspiração na programação orientada por aspectos, a natureza dos conceitos abordados nas definições semânticas é explorada na criação de módulos. Consequentemente, atinge-se um alto nível de separação de interesses ao se evitar o entrelaçamento de conceitos expressos nas equações semânticas. O uso de componentes evita o espalhamento do contexto e garantem a separação de interesses ortogonais. Além disso, ao contrário da transformação de denotações, a combinação de componentes preserva o raciocínio equacional.

Assim como a técnica apresentada, a semântica de mônadas (Moggi, 1989, 1991) aborda o problema de modularidade de definições semânticas ao abstrair o contexto das equações semânticas. No entanto, mônadas são mecanismos de abstração complexos. Consequentemente, as definições semânticas monádicas atingem um elevado nível de modularidade, mas dependem de complexas operações de transformação de mônadas. A técnica apresentada é mais simples, explorando a padronização de interfaces das denotações e o encapsulamento de conceitos fundamentais em componentes.

É fato que a semântica monádica permite a definição incremental do modelo semântico. Pode-se por exemplo definir primeiro um núcleo simples, acrescentando depois continuação, *environment* e *store*. Entretanto, o custo dessa flexibilidade se faz visível na complexidade das operações de transformação de mônadas. A semântica denotacional clássica, por outro lado, é mais simples mas menos flexível. A técnica apresentada é mais simples pois recursos desnecessários não se fazem presentes nas equações semânticas por omissão dos componentes relacionados.

A semântica baseada em componentes (Mosses, 2005) é a principal inspiração deste trabalho. No entanto, Mosses utiliza abordagens operacionais, semântica de ações e semântica operacional implicitamente modular, como base para a definição da semântica de seus componentes. Como sugerido por Mosses (2005), a baixa legibilidade e escalabilidade decorrente do uso explícito de informações de contexto nas denotações é um obstáculo para o uso da semântica denotacional como base de uma técnica de definição por meio de componentes. Esse obstáculo é transposto por meio do uso de uma biblioteca de componentes, a qual encapsula o fluxo de informações de contexto entre as denotações.

6 Considerações Finais

Este trabalho apresenta uma nova abordagem de semântica denotacional baseada em componentes. Essa técnica utiliza uma biblioteca de componentes reutilizáveis que encapsulam conceitos fundamentais e recorrentes das linguagens de programação imperativas e podem ser usados para combinar denotações e compor a semântica de construções mais elaboradas. Essa abordagem é inspirada no trabalho de Mosses (2005), o qual aborda a definição de componentes para a semântica de ações e para a semântica operacional implicitamente modular.

A biblioteca de componentes permite a remoção da dependência aparente das informações de contexto na redação das equações semânticas, tornando-as mais modulares. Para isso, as assinaturas dos componentes são padronizadas e o fluxo de informações de contexto é encapsulado nos componentes. Consequentemente, as definições semânticas mapeiam as construções da sintaxe abstrata das linguagens de programação diretamente a combinações de componentes denotacionais. Como resultado, as definições semânticas se tornam mais legíveis.

O encapsulamento de conceitos fundamentais e recorrentes das linguagens de programação imperativas em componentes facilita a definição semântica de linguagens imperativas de grande porte. Isso ocorre porque grande parte dos conceitos abordados por essas definições são modeláveis por componentes previamente definidos. O exercício de definição se resume então a combinar os componentes de forma a modelar a semântica desejada. Assim, eleva-se o nível de abstração das definições semânticas. O objetivo principal dessa abordagem é melhorar a legibilidade e a escalabilidade da semântica denotacional.

No entanto, a biblioteca de componentes é essencialmente um trabalho em aberto. Conforme as linguagens de programação evoluem, novos conceitos precisam ser incorporados. A abordagem recomendada nesse caso é o encapsulamento desses conceitos em novos componentes. Não obstante, caso maior liberdade se faça necessária, é também possível utilizar semântica denotacional clássica em conjunto com os componentes na redação de equações semânticas. No pior caso, a incorporação de novos componentes pode conflitar com os mecanismos já previstos pela biblioteca e, potencialmente, exigir a restruturação de um número muito grande de componentes pré-existentes.

Entretanto, é possível facilitar a extensão da biblioteca por meio da definição dos componentes por meio de outras abordagens denotacionais como a semântica monádica. Nesse caso, a complexidade da nova definição fica encapsulada nos componentes, logo ganha-se maior flexibilidade sem que a legibilidade ou a escalabilidade diminuam. Essa é uma forma de aliar uma metodologia já existente à metodologia proposta. Por isso, este trabalho complementa as contribuições de trabalhos anteriores. Com isso, acredita-se que este trabalho contribui para a popularização da semântica formal de linguagens de programação.

A biblioteca de componentes de semântica denotacional apresentada neste trabalho foi implementada na linguagem Haskell. Unindo o sistema implementado por meio da biblioteca a um *parser* que crie a árvore de sintaxe abstrata dos programas, é possível criar um protótipo para validar definições semânticas e realizar experimentos com extensões de linguagens.

Finalmente, como trabalhos futuros podem-se citar a incorporação de novos componentes à biblioteca e a inserção da técnica apresentada no desenvolvimento de compiladores. O primeiro tornaria a biblioteca mais abrangente e facilitaria a definição semântica de novas linguagens, enquanto o segundo aumentaria a confiabilidade dos compiladores ao aproximar especificação e implementação. Uma forma de se criar essa sinergia é a geração de código a partir de definições semânticas baseadas em componentes. Isso permitiria a inserção da semântica formal no processo de compilação junto dos já bem estabelecidos analisadores léxico e sintático, *backend* e otimizações de código.

Referências

Curry, H.B., Feys, R.: Combinatory Logic I. North-Holland, Amsterdam (1958)

Gordon, M.J.C.: The denotational description of programming languages - an introduction. Springer-Verlag (1979)

Hughes, R.J.M.: Super-combinators a new implementation method for applicative languages. In: Proceedings of the 1982 ACM symposium on LISP and functional programming. pp. 1–10. LFP '82, ACM, New York, NY, USA (1982)

Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1995)

Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in computer science. pp. 14–23. IEEE Press, Piscataway, NJ, USA (1989)

Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991)

Mosses, P.D.: The modularity of action semantics. Internal Report IR–75, Dept. of Computer Science, Univ. of Aarhus (1988), revised version of a paper presented at a CSLI Workshop on Semantic Issues in Human and Computer Languages, Half Moon Bay, California, March 1987 (proceedings unpublished)

Mosses, P.D.: The varieties of programming language semantics. In: Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics: Akademgorodok, Novosibirsk, Russia. PSI '02, vol. 2244, pp. 165–190. Springer-Verlag, London, UK, UK (2001)

Mosses, P.D.: A constructive approach to language definition. Journal of Universal Computer Science 11(7), 1117–1134 (2005)

Santos, G.H.S.: Component based denotational semantics (Mar 2013a), `https://code.google.com/p/denotational-semantics-component-library/`

Santos, G.H.S.: Semântica denotacional escalável de linguagens de imperativas. Master's thesis, UFMG (2013b)

Tirelo, F., Bigonha, R.S., Saraiva, J.: Disentangling denotational semantics definitions. Journal of Universal Computer Science 14(21), 3592–3607 (dec 2008)

Zhang, Y., Xu, B.: A survey of semantic description frameworks for programming languages. SIGPLAN Not. 39, 14–30 (March 2004)